Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 141 (2005) 103–119

www.elsevier.com/locate/entcs

Quantifying the Benefits of SSA-Based Mobile Code

Wolfram Amme[1]

Institut für Informatik
Friedrich-Schiller-Universität Jena
Jena, Germany

Jeffery von Ronne[2] Michael Franz[3]

Information and Computer Science
University of California, Irvine
Irvine, CA, United States

Abstract

High-performance just-in-time compilers for Java need to invest considerable effort before actual code generation can commence. This is in part due to the very nature of the Java Virtual Machine, which is not well matched to the requirements of optimizing code generators. Alternative transportation formats based on Static Single Assignment form should theoretically be superior to virtual machines, but this claim has not previously been validated in practice. This paper revisits the topic and attempts to quantify the effect of using an SSA-based mobile code representation (IR) instead of a virtual-machine based one.

To this end, we have integrated full support for a verifiable SSA-based IR into Jikes RVM, an existing Java execution environment. The resulting system is capable of loading and executing Java programs represented in either format, traditional JVM bytecode as well as the SSA-based representation, and it can even execute programs made up of a mixture of the two formats. In our implementation, the two alternative just-in-time compilation pipelines share a common low-level code generator.

Performance results are encouraging and show simultaneous improvements in both compilation time and code quality relative to Jikes RVM's standard optimizing compiler for JVM class files. They support the hypothesis that SSA-based intermediate representations offer advantages in the context of just-in-time compilation.

Keywords: virtual machine, single static assignment form, SafeTSA

doi:10.1016/j.entcs.2005.02.046

1 Introduction

In 2001, we described SafeTSA, a type- and reference-safe representation for Java that is based on static single assignment (SSA) form [3]. SafeTSA was explicitly designed to replace the Java Virtual Machine bytecode language (JVML), and argued that such an SSA-based representation should provide significant advantages over virtual-machine based formats when shipping mobile code, particularly in the context of eventual just-in-time (JIT) compilation. However, benchmarks presented in the paper related only to the size of the resulting representation (which was denser than JVML), not to its JIT compilation or execution performance.

In this paper, we report on how we have taken the SafeTSA format and integrated it into IBM's Jikes RVM [1] virtual machine for the PowerPC architecture. The result is a Java execution environment that is capable of processing both Java class files and SafeTSA files interchangeably. It can even execute programs in which some of the classes have been compiled into Java class files and others into SafeTSA files, which are then all combined during dynamic loading.

In an attempt to quantify the relative merits of JVML vs. SafeTSA in the context of JIT compilation (and by extension, the relative merits of stack-based vs. SSA-based intermediate representations for mobile code in general), we compiled a series of benchmark programs into both Java class files and SafeTSA files. These were then used as inputs for Jikes RVM's standard JVML optimizing compiler and for the SafeTSA compiler, permitting a direct comparison of the respective code-generation times required by the two compilers as well as the performance of the resulting native code. On the basis of these measurements, we assess whether an SSA-based intermediate representation format can serve as a feasible replacement for JVML.

In the following sections, we first introduce some of the key features of the SafeTSA format. We then give a brief overview of the Jikes RVM system, particularly its code generator and internal data structures. Following this, we describe the implementation of our SafeTSA compiler and its integration into the Jikes RVM system. This is followed by a discussion of the benchmark results, reporting on both code-generation time and on the generated code's performance. The paper concludes with a summary and discussion of future work.

[1] Email: amme@informatik.uni-jena.de
[2] Email: jronne@ics.uci.edu
[3] Email: franz@uci.edu

```
public class A {
  int f1;
  int f2;
}

public class B {
  static in foo (A a) {
    int sum = a.f1;
    int i = 1;
    while (i < 10) {
      sum += sum;
      i++;
    }
    return sum * a.f2;
  }
}
```

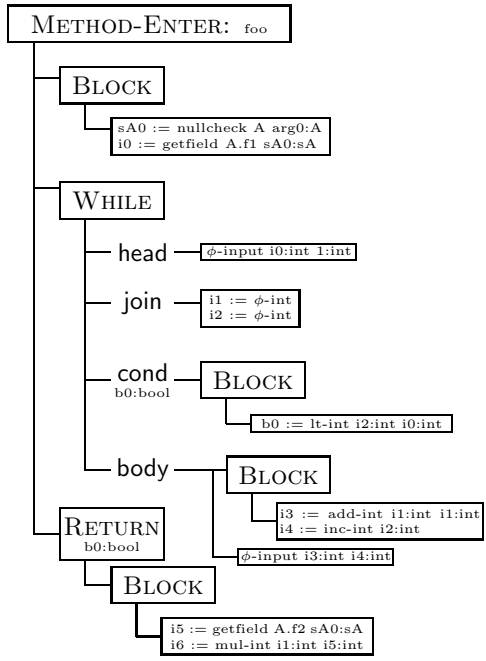


Fig. 1. Example program and its representation in SafeTSA

2 The SafeTSA Representation

SafeTSA[4] is an intermediate representation designed to be target-machine independent, simple to verify, and easy to translate into optimized native code. SafeTSA achieves this through a novel combination of several key features: an embedded control structure tree, instructions in SSA form, dominator-based referential integrity, type safety through type separation, a type system extended to support key optimizations, and a carefully chosen instruction set.

Figure 1 contains a graphical representation of the SafeTSA file produced from the source program given in the figure. It will be referred to in the following discussion of SafeTSA's key features. Rather than allowing arbitrary branch instructions, SafeTSA conveys the program's control flow through a tree of high-level control structure elements closely paralleling those of the Java source language. This *control structure tree* can be seen depicted as the boxes and the connecting lines in Figure 1; as a first approximation, the control structure tree can be thought of as a method's abstract syntax tree with its expressions removed. The use of control structure trees restricts SafeTSA methods' control flow graphs to a well defined subset of reducible control flow graphs. This simplifies the machine-specific code generation and optimization

[4] The name SafeTSA stands for *Safe Typed Single Static Assignment Form* and predates the formation of the U.S. Transportation Security Administration.

as well as dominator tree derivation [7].

Static single assignment form [6] is a standard representation for optimizing compilers. SafeTSA is based on SSA form, leveraging its benefits during the JIT compilation phase but shifting off-line the costs of producing SSA form. The key feature of SSA form is that each 'variable' that is used in the program's SSA representation is defined at a single location in the representation. This can be seen depicted in Figure 1 as the unique variable on the left-hand side of each instruction.

To handle merge points of values originating at different points in the control flow, SSA provides special ϕ-functions which select among alternative input variables at runtime based on the CFG edge by which the execution reached the join node containing the ϕ-functions. As can be seen in Figure 1, SafeTSA has a separate "ϕ-input" at each location where control can be transfered to the join node; each "ϕ-input" has as many operands as there are ϕ-instructions in the join node.

An important property of a correct SSA program representation is that, for all instructions A and B, if A uses the result variable of B, A must be dominated by B.[5] This property, *referential integrity*, is the necessary and sufficient condition that all variables in an SSA program are defined before they are used on every possible path through the CFG. The serialized SafeTSA encoding enforces this property statically by referencing the input variables of an operation using a relative addressing scheme [3]. Because this relative addressing changes for every instruction, it is not depicted in Figure 1.

SafeTSA simplifies type checking through *type separation* and explicit cast operations. Object oriented source languages will normally allow a subtype to be used anywhere the parent type is used. In contrast, SafeTSA maintains a separate name space for variables of each type: every operation that defines a result variable defines that variable to be of a particular type, and every operation that uses a variable can only refer to variables of the correct input type. Type separation is depicted in Figure 1 in the naming convention of the SSA output variables, and in the "$:type$" notation on the input variables; type separation requires that input and output variable types match each instruction's type signature exactly. If a subtype must be used as a type, an explicit cast is placed in the program representation through an instruction that takes the subtype variable as input and produces the type as its output variable. If, when the JIT compiler processes the method, this cast can be

[5] For ϕ-instructions, the use of the input variable is considered to occur at the end of each of the blocks that precede the join node in the CFG. The output variable defined by the ϕ-instruction(s), however, is considered to occur at the top of its join node. In SafeTSA, this is implemented through ϕ-inputs that are separated from the ϕ at the join node.

verified to always be correct (by consulting the virtual machine's class hierarchy), then it will produce no executable code, but if the correctness cannot be guarantied (e.g., a cast from a type to a subtype), a dynamic check will be necessary. This combination of SSA form and syntactic type separation is trivial to verify, but it, along with referential integrity, replaces the complex stack-based type-inference required by Java bytecode verification.

The type system of SafeTSA is, at its core, the same as that of Java and Java bytecode. It allows the same types of objects in the garbage collected virtual machine's heap, and the SSA variable types may be any of the Java primitive types (int, float, etc.) or a reference type restricted to instances of a particular class type, a particular interface type, or a particular array type, according to the same rules governing Java reference types. But the SafeTSA *type system* has also been *extended to support optimization.*

In particular, for each Java reference type, SafeTSA adds a 'safe' reference type that can only be produced by a null-check operation. All operations that act on the heap object require the null-checked 'safe' reference type as input. As a consequence, the null-check can be safely separated from the access using the null-checked reference, allowing some classes of redundant null-checks to be optimized away when the SafeTSA representation is produced. An example of this can be seen in Figure 1, where the first getfield requires a null-check to convert the argument from type 'A' to 'sA', but the second is able to use the 'sA0' variable, which is already known to be safe. Similarly, separate types are added to represent the results of bounds-checked array element address computations. The introduction of safe reference and array element types allows the elimination of redundant null- and index-checks when performing simple common subexpression elimination [2].

While the SSA representation's type safety is preserved through type separation, the *instruction set* of SafeTSA was carefully *chosen to maintain the type and memory safety of the heap space.* The SafeTSA instruction set can be divided into two main classes. The first class contains those operations that are functional (i.e., take zero or more inputs and produce an output based solely on those inputs); this includes primitive computation (e.g., integer add), casts, and checks; the effects of these instructions are entirely captured the SSA model. The second class of instructions contain those that can alter the virtual machine in other ways (e.g., by modifying the garbage collected heap space). This class includes both field and array manipulation, as well as method and constructor invocation. The getfield in Figure 1 is an example of this second class. These operations closely follow the semantics of their JVML counterparts and enforce the same type and memory safety invariants.

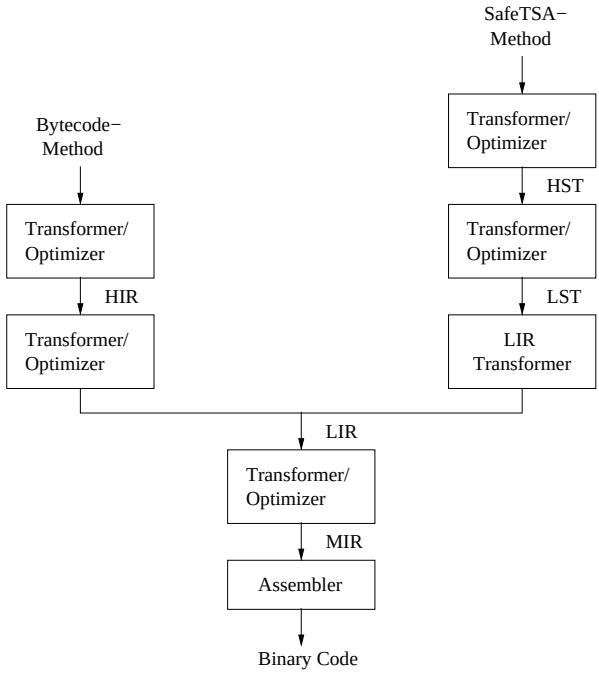


Fig. 2. Compiling JVML and SafeTSA methods in Jikes RVM

3 Overview of Jikes RVM

Jikes RVM is a Java Virtual Machine developed at IBM Research [4]. Jikes RVM possesses several unique features, two of which are particularly relevant to the work presented here: it is compile-only (i.e., it has no interpreter), and it is written almost entirely in Java.

Instead of having an interpreter, Jikes RVM features multiple JVML to native code compilers. One is the 'Baseline' compiler, which exists for debugging and verification purposes; it produces native code that directly implements JVML's stack model as closely as possible and is in many ways comparable to an interpreter. Jikes RVM also has an 'Optimizing' compiler [5], which consists of multiple phases and can be operated at various levels of optimization.

The phases of the Jikes RVM optimizing compiler communicate through a series of intermediate representations: a high-level intermediate representation (HIR), a low-level intermediate representation (LIR), and a machine-specific intermediate representation (MIR), as can be seen in Figure 2. A JVML method is initially translated into HIR, which can be thought of as a register-oriented transliteration of the stack-oriented JVML. The LIR differs from the HIR in that certain JVML instructions are replaced with Jikes RVM-specific implementations (e.g., an HIR instruction to read a value from a field would be expanded to LIR instructions that calculate the field address and then perform

a load on that address). The lowering from LIR to MIR renders the program in the vocabulary of the target instruction set architecture. The final stage of compilation is to produce native machine code from the method's MIR. Depending on the configuration of the optimizing compiler, optimizations can be performed in each of these IRs.

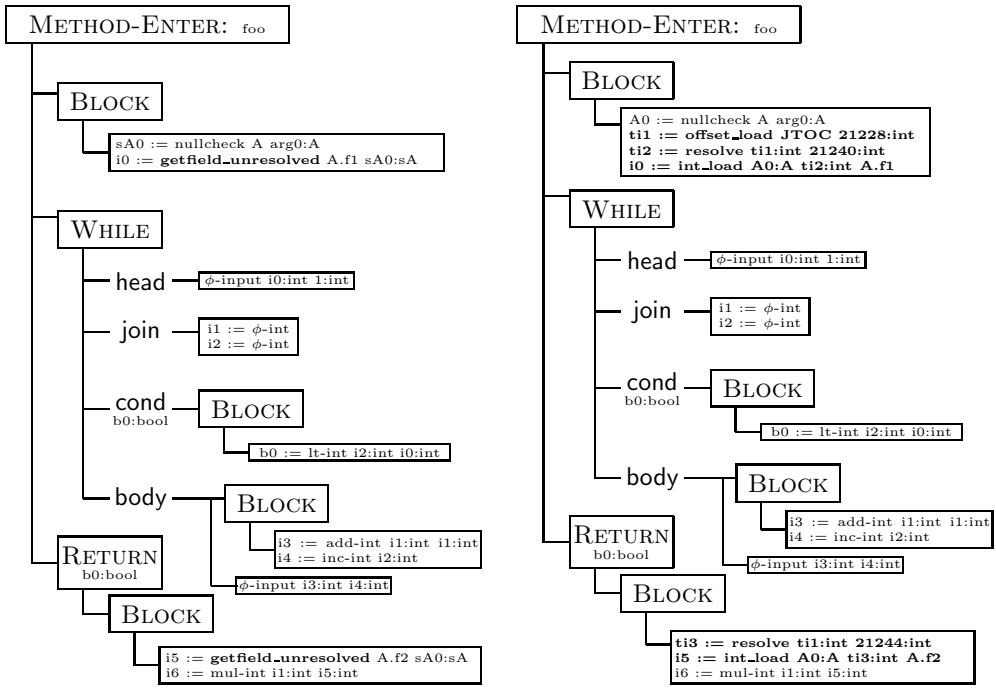
Because Jikes RVM is written in Java, the key data structures are accessible to the VM as Java arrays. The most important of these is the Jikes RVM's table of contents (JTOC). The JTOC is an array containing (or containing references to) all globally-accessible entities (i.e., all of the constants, each type's type information block (TIB), meta-objects for static fields and methods, etc. can be found as an index in the JTOC). Loading a class consists of instantiating the appropriate TIB and meta-objects and adding them to the JTOC or the class's TIB.

Methods are compiled the first time they are invoked. When the compiler finds references to fields or methods that have not yet been loaded, it includes code to resolve the field or method just before using the field or method the first time. Jikes RVM facilitates dynamic class loading by maintaining two offset tables, OffsetTableField and OffsetTableMethod. There is an entry in one of these tables for every known field and method. When resolution of a field or method is required, the appropriate entry is accessed. If it is valid, the offset is used to calculate an address. If it is not valid, the class loader will be called to load the appropriate class and write a valid offset into the appropriate offset tables.

One further feature of the Jikes RVM system is that null-check instructions, which are explicit in the intermediate representations of Jikes RVM's optimizing compiler, can be replaced by hardware checks. To reduce the number of null-check instructions that have to executed at runtime, a so-called null-check combiner checks during the generation of MIR for each null-check instruction if the instruction can be eliminated and combined with a directly following load or store instruction. References for which an explicit null-check instruction has been eliminated in this way, will still be null-checked at the time of memory access, because the execution of a load or store instruction with base address *null* will throw a hardware interrupt that is trapped by the Jikes RVM system.

4 Integrating SafeTSA

By integrating SafeTSA class loading and a SafeTSA compiler into the Jikes RVM system, we have built a VM that can execute both SafeTSA- and JVML-compiled Java programs. Thus, functionally, it does not matter whether the



(a) High Level SafeTSA (b) Low Level SafeTSA

Fig. 3. HST and LST representation of method foo().

whole program has been compiled to SafeTSA, or if it exists as JVML class files, or if the program is provided as a heterogeneous mix of both SafeTSA and JVML classes.

With respect to dynamic class loading, the modified Jikes RVM system treats SafeTSA classes in a manner analogous to traditional JVML classes. This was accomplished by modifying the class loader so that whenever it loads a new class, it will check the class file repositories for SafeTSA classes. Whenever the modified class loader finds a SafeTSA file, it will load the SafeTSA file and set up the necessary JTOC and TIB entries; method invocations on classes loaded from SafeTSA files result in the SafeTSA compiler being invoked to produce executable code. If no SafeTSA file exists, the class loader simply loads the appropriate Java class file, and any method invocation on the JVML class will result in the standard JVML optimizing compiler being invoked to compile the method.

Figure 2 shows the internal structure of our SafeTSA compiler. Initially, the compiler transforms the method into its high-level SafeTSA representation (HST). An HST representation of a SafeTSA method is an intermediate representation that is largely independent of the host runtime environment but differs from the original SafeTSA method in that there is some resolution of

accessed fields and methods. Next, the SafeTSA method is transformed from HST into the low-level SafeTSA representation (LST). This process expands some HST instructions into a host-JVM specific LST operations, specializing them for Jikes RVM's object layout and parameter passing mechanisms. After this transformation the LST method is optimized and transformed into the same LIR used by Jikes RVM's JVML optimizing compiler. The JVML compiler's LIR to MIR phase is used unmodified to perform instruction selection, scheduling, and register allocation.

A consequence of Java's dynamic class loading is that a method may refer to fields, methods, and types of classes whose implementation has not yet been loaded into the VM. In this case the JIT compiler will be unable to resolve the access at compile time. Instead of directly accessing the data structure, the JIT compiler must insert special "resolve" instructions that cause the implementation to be loaded and the appropriate VM data structures instantiated. The SafeTSA compiler inserts these resolution instructions during the creation of the HST IR. This is in fact, the main difference between the serialized SafeTSA representation and HST: a getfield_unresolved setfield_unresolved or call_unresolved will be substituted for each getfield/setfield or call instruction of the SafeTSA representation that operates on a class which is not yet loaded. Figure 3(a) shows what the HST for the method foo would look like if class A has not been loaded prior to foo's compilation. As is evidenced by the getfield_unresolved instructions.

Once the method is in HST, it can be lowered to LST by expanding certain high-level instructions to Jikes RVM specific implementations. Mostly, this consists of performing address computations using offsets from Jikes RVM's JTOC and TIBs. It also involves translating SafeTSA check and cast operations to their Jikes RVM equivalents, materializing constant operands, and translating high-level storage accesses into low-level load and store instructions. Figure 3 shows the optimized LST that would be created from the example program. In the LST, safe types have been converted back into normal Jikes RVM types and the high-level getfield_unresolved instructions have been lowered to sequences of instructions that perform resolution and accesses the field: An offset_load instruction delivers the address of the offset table OffsetTableField. The resolve instruction checks attempts to load the offset based on the field's field dictionary index (in the example program, a.f1 has the field dictionary entry 21240 and a.f2 the entry 21244). A final low-level int_load instruction is used to actually load the field once the offset has been determined. Because of common subexpression elimination, the second getfield_unresolved does not require an offset_load instruction when translated to LST.

The translation from LST to LIR is the final phase of the SafeTSA compiler

```
 1 LABEL0
 2 prologue l0pi(A,x,d) =
 3 LABEL1
 4 null_check t2v = l0pi(A,x,d)
 5 int_load t3i([I) = JTOC(int), 21228
 6 LABEL3
 7 int_load t4i(int) = t3i([I), 21240
 8 int_ifcmp t4i(int), 0, ! =, LABEL2
 9 LABEL4
10 resolve A.f1
11 goto LABEL3
12 LABEL2
13 int_load t6i(int) = l0pi(A,x,d), t4i(int), A.f1, t2v
14 int_move t7i(int) = 1
15 goto LABEL6
16 LABEL7
17 int_add t6i(int) = t6i(int), t6i(int)
18 int_add t7i(int) = t7i(int), 1
19 LABEL6
20 int_ifcmp t10v = t7i(int), 10, <, LABEL7
21 LABEL8
22 int_load t4i(int) = t3i([I), 21244
23 int_ifcmp t4i(int), 0, !=, LABEL10
24 LABEL9
25 resolve A.f2
26 goto LABEL8
27 LABEL10
28 int_load t11i(int) = l0pi(A,x,d), t4i(int), A.f1, t2v
29 int_mul t12i(int) = t6i(int), t11i(int)
30 return t12i(int)
```

Fig. 4. LIR representation of method foo().

and is composed of three main tasks: the translation of the control structure tree into branch instructions, straightforward translation of LST instructions into LIR instructions, and translation from SSA variables into LIR's virtual registers. Figure 4 shows our example program translated into LIR, which consists of 9 basic blocks. Instructions 5–11 resolve 'a.f1'; instructions 21–26 resolve 'a.f2'; instructions 15–20 represent the while-loop.

During the translation of SSA values to virtual registers, each of the ϕ-functions must normally be translated into a move at each of the ϕ-functions' ϕ-inputs. This move transfers the contents of the appropriate ϕ-input operand into a result register correspond to the ϕ instruction. Therefore, normally, when generating the LIR for method foo it is necessary to insert four move instructions (two instructions at the of the block before the while loop and two instructions at the end of the block that represents the loop body) into the intermediate representation.

A naive resolution of ϕ-functions will, however, often produce more moves than is required, because a more sophisticated resolution could coalesce the virtual register for some ϕ-instructions with one or both of their input registers. To reduce the number of move instructions that have to be inserted into the LIR, the SafeTSA compiler can be instructed to perform a ϕ-move optimization. In doing so, the SafeTSA compiler checks before each insertion of a move instruction if the target register can be coalesced with the source register of the instruction. If the coalescing of a target and source register is feasible, both registers will be coalesced into a single register instead of inserting a move instruction into the LIR. As the LIR of Figure 4 shows, the number of move instruction inserted into the example program can be reduced to one through the use of this ϕ-move optimization.

Name	Description	Size (bytes)
Section 2		
Crypt	IDEA encryption	19875
HeapSort	Integer sorting	7535
LUFact	LU Factorization	18000
SOR	Successive over-relaxation	2541
SparseMat	Sparse Matrix multiplication	6269
Section 3		
Euler	Computational Fluid Dynamics	38320
Moldyn	Molecular Dynamics simulation	14921
MonteCarlo	Monte Carlo simulation	117445
RayTracer	3D Ray Tracer	40173
Search	Alpha-beta pruned search	21582

Table 1
Benchmark programs

	JVML	S	SP	SS	SPS	SLPS	SCLPS
Sun javac	✓						
SafeTSA		✓	✓	✓	✓	✓	✓
common subexpression elimination							✓
deadcode elimination						✓	✓
constant propagation			✓		✓	✓	✓
ϕ-move optimization				✓	✓	✓	✓

Table 2
Mobile-Code Formats and Optimizations

5 Results

SafeTSA provides a mechanism for the safe transport of optimized code, but in order to empirically assess whether SafeTSA delivers the expected performance benefits, we ran a series of benchmarks through the Jikes RVM system in which we compared the execution time and required compilation time of programs compiled to JVML and SafeTSA.[6] All results discussed in the following were obtained by running the benchmark programs from sections 2 and 3 of the Java Grande Forum Sequential Benchmarks (JGF) [8]. The Java Grande Benchmarks were chosen because they were freely available in source code and seemed appropriate for measuring compilation time and the performance of generated code. Table 1 lists these benchmark programs and provides a short description and its program size.

Table 2 shows the mobile-code formats and optimizations used during

[6] All reported execution times also include the time required for the JIT compilation of the benchmark program's methods.

Benchmark	**Time in seconds**				
	JVML	SafeTSA (SP)	Δ%	SafeTSA (SPS)	Δ %
Crypt	5.498	5.370	-2.33	5.392	-1.93
HeapSort	3.084	3.063	-0.68	3.051	-1.07
LUFact	3.320	3.247	-2.20	3.232	-2.65
SOR	10.420	10.388	-0.36	10.377	-0.47
SparseMat	23.266	23.040	-0.97	23.019	-1.06
Euler	59.671	58.132	-2.58	58.154	-2.54
Moldyn	14.785	14.642	-0.97	14.641	-0.97
MonteCarlo	38.175	37.580	-1.56	37.287	-2.33
RayTracer	55.932	56.597	1.19	56.476	0.97
Search	20.067	20.322	1.27	20.324	1.28

Table 3
Execution Time: JVML versus SafeTSA

benchmarking. JVML is used to denote Java class files produced using version 1.2.2 of Sun javac. The SafeTSA files were produced with various optimizations enabled during the compilation of Java source code into the SafeTSA format. In addition, in some cases, ϕ-move optimization was used during the generation of LIR code. In the following discussion, we will refer to SafeTSA files that have been produced with constant propagation (SP) as their only optimization as 'baseline' SafeTSA files.

All results were obtained on a PowerMac with a 733 MHz PowerPC G4 processor running Mandrake Linux 7.1. The system has 1.5 GB of main memory and a 256KB L2 Cache. The Jikes RVM system that we used for our measurements was generated from a modified version of Jikes RVM 2.2.0 using the FullOptNoGC option (i.e., the bootimage containing the Jikes RVM JVM was itself produced by the Jikes RVM optimizing compiler, and all JVM classes were included in the bootimage). In order to get stable results from one run to the next, we disabled garbage collection and ran the system in the single user mode. We ran each benchmark several times and took the best result, but in this stable configuration, all of the overall execution times varied by less than 0.005s, and in most cases, the variation was less than we could measure.

In benchmarking, we were mainly interested in examining the relative merits of JVML and SafeTSA's high-level designs when they are used as an intermediate representation for JIT compilation. Therefore, all measurements have been performed running both the SafeTSA compiler and Jikes RVM's optimizing bytecode compiler in optimization level 0. When using this optimization

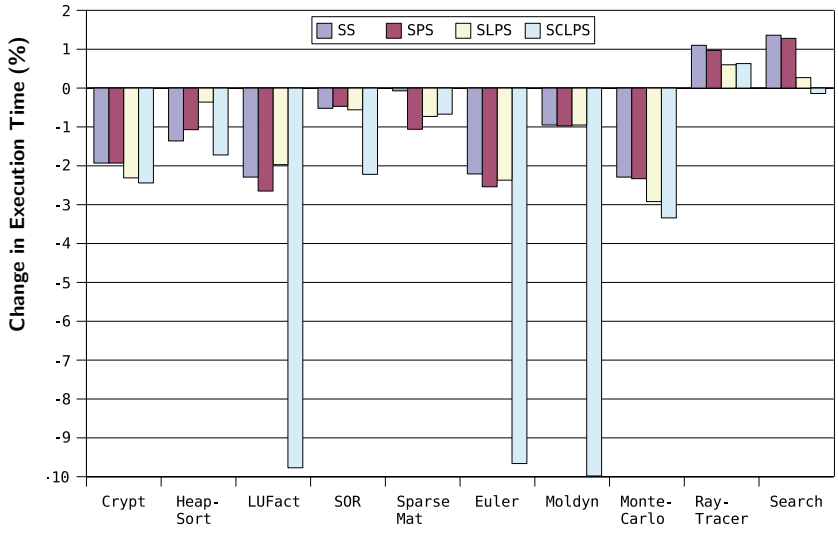


Fig. 5. Influence of static optimizations on the execution times

level, the SafeTSA compiler supports utilizes no additonal optimizations except for ϕ-move optimization and the Jikes RVM's bytecode compiler only performs optimizations which are required for an accurate bytecode compilation. Because the SafeTSA compiler does not yet support inlining between JVML and SafeTSA methods we globally disabled the inlining of method calls.

Table 3 compares the execution times in seconds for the benchmark programs compiled with Jikes RVM's JVML JIT to the runtime that can be achieved when using baseline SafeTSA files as input for the SafeTSA compiler. Both with and without ϕ-move optimization, the SafeTSA-based version outperformed the JVML version in 8 out of 10 benchmarks. At a first glance the performance gains (between 0.36% to 2.58% reduction in runtime) appear somewhat unimpressive, but it should be noted that baseline SafeTSA files differ from JVML files simply in the non-existence of assignments to local variables. As the results of Table 3 indicate, in most cases, ϕ-move optimization reduces the execution time but only modestly. However, our comparable studies using the simpler programs in Section 1 of the JGF Sequential Benchmarks indicate that, in some cases, the runtime of programs can be improved up to 8% when performing ϕ-move optimization.

To ascertain the influence that statically-performed ahead-of-time optimizations have on the execution time of the benchmark programs, a series of differently optimized SafeTSA files were generated and subsequently compared with respect to execution time. Figure 5 depicts the measured runtimes for different SafeTSA versions (SS, SPS, SLPS and SCLPS) relative to the

Benchmark	Instructions			Nullchecks			Indexchecks		
	Before	After	Δ%	Before	After	Δ%	Before	After	Δ%
Crypt	721	681	-5.55	86	63	-26.74	81	81	±0
HeapSort	194	185	-4.64	24	24	±0	21	21	±0
LUFact	848	770	-9.20	94	69	-26.60	96	81	-15.63
SOR	181	165	-8.84	23	15	-34.78	23	22	-4.35
SparseMat	237	231	-2.53	29	28	-3.45	32	32	±0
Euler	7296	6698	-8.20	1666	1422	-14.65	1230	1103	-10.33
Moldyn	1302	1293	-0.69	108	108	±0	49	49	±0
MonteCarlo	1878	1773	-5.59	181	140	-22.65	50	46	-8.01
RayTracer	1340	1177	-12.16	203	121	-40.39	14	14	±0
Search	1580	1486	-5.95	111	99	-10.81	283	274	-3.18

Table 4
Number of instructions, null-checks and bounds-checks

runtimes that could be achieved during the execution of its JVML based counterparts. The measurements show that constant propagation and dead code elimination tend to result in rather minor runtime improvements. Moreover, in some cases the application of these optimizations can lead to performance degradation, which is probably caused by reduced program locality. Whereas, except for one benchmark program, the application of common subexpression elimination consistently resulted in considerable performance gains. Overall, the SafeTSA files produced with all three optimizations failed to outperform ther JVML counterparts in only one benchmark program. In all other cases, fully optimized SafeTSA files outperformed JVML files sometimes considerably, and for three benchmark programs, the execution times measured for SafeTSA files were more than 9% less than the times required for JVML files.

For a better understanding of the runtime results that we observed for optimized SafeTSA, further investigations were performed to determine what kind of subexpression elimination was responsible for the achieved performance gains. Table 4 shows the number of null-checks and bounds-checks that could be eliminated from the SafeTSA files during the three optimization phases. As the table shows, 7% of all instructions, 17% of all null-checks, and 8% of all bounds-checks could be eliminated from the programs. A further review of the program sources showed that the elimination of superfluous bounds-checks is the main reason that lead to the improved runtime behavior of optimized SafeTSA files. As an example, the elimination of 15.63% and 10.33% of the bounds-checks in the benchmark programs LUFact and Euler resulted in 9.77% and 9.66% shorter execution times. In contrast, the 9.98% reduction in the execution time of the Moldyn benchmark was caused by the elimination of unnecessary floating point operations that must execute several times inside

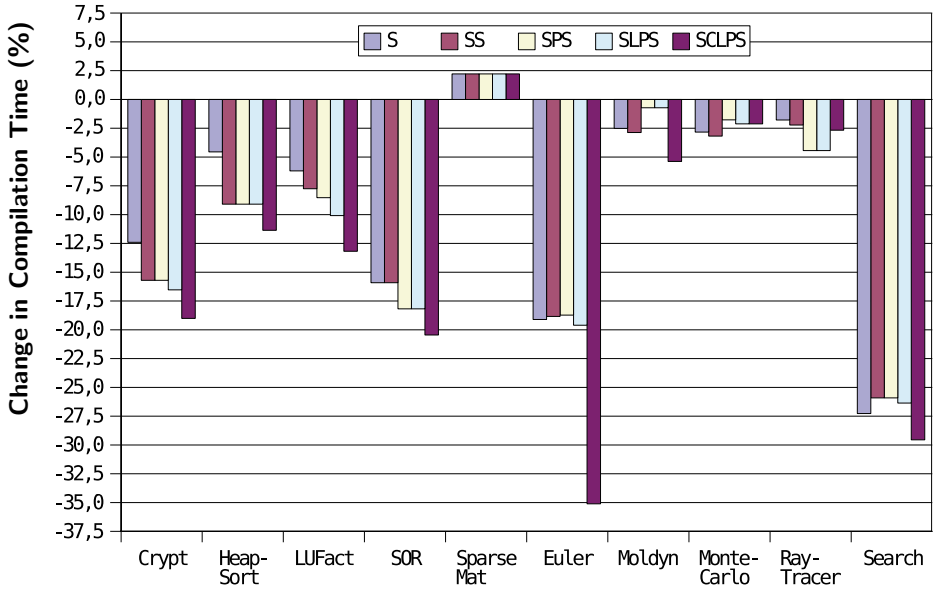


Fig. 6. Overall compilation time (relative)

a loop.

Even though the target code produced from SafeTSA is often better than its JVML counterpart, the measured compilation times for both compilers indicate that, in most cases, a SafeTSA compilation needs less time than a JVML-based compilation. For our benchmarks, the compilation times made up only 1-4% of the total execution time. The largest compilation time was the 1.6 seconds required by the JVM bytecode compiler for *Euler* (which also had the largest total execution time of 59.67s); all of the other benchmarks had compilation times of less than 0.3s.

Figure 6 contains the compilation time for different optimized SafeTSA files relative to a JVML-based compilation for all benchmark programs. For baseline SafeTSA, compilation was up to 27% faster than the compilation of JVML, and compilation of fully optimized SafeTSA was up to 35% faster. The reduced compilation times of fully optimized SafeTSA can be explained by the decreased program size of optimized SafeTSA files.

6 Summary and Future Work

We have described the integration of SafeTSA support into Jikes RVM, an existing dynamic optimization system, creating a complete runtime environment that supports a heterogeneous mix of SafeTSA and JVML class files. We have used this system to run the Java Grande Benchmarks, in an attempt to assess

the relative merits of SafeTSA vs. JVML as inputs to a JIT compiler. The results from these benchmark runs show that SafeTSA can produce better performing native code with a shorter compilation time.

It should also be pointed out that these numbers do not include verification time, since Jikes RVM 2.2.0 does not verify class files. SafeTSA verification is much simpler than JVML verification, so that including verification into the total time needed from download to execution would tilt the balance considerably further in SafeTSA's favor.

While the results so far are impressive, they are still preliminary and very probably under-represent the benefits of SafeTSA. Current plans for improving the compiler include writing an SSA aware register allocator and final machine code generator, which will take advantage of the fact that Jikes RVM's LST is also in SSA form. We also plan to in investigate the possibility of incorporating program annotations into an SSA-based representation that would improve code quality and/or reduce compilation time, while still retaining safety.

Acknowledgement

We wish to thank Fermín Reig, Andreas Hartmann, Ning Wang, and the various anonymous referees for their comments on earlier versions of this paper. We would also like to thank everyone who has contributed to the SafeTSA project in some way, especially Philipp Adler, Alexander Apel, Hartmut Arlt, Niall Dalton, Andreas Hartmann, Thomas Heinze, Timo Mai, Andre Möller, Jan Peterson, Prashant Saraswat, and Ning Wang. In addition, we wish to extend our thanks to Michael Hind and the other contributors to the Jikes Research Virtual Machine at IBM Research and elsewhere.

This investigation has been supported in part by the Deutsche Forschungsgemeinschaft (DFG) under grants AM-150/1-1 and AM-150/1-3, by the National Science Foundation (NSF) under grant CCR-9901689, and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Lab oratory (AFRL) under agreement number F30602-99-1-0536.

References

[1] Alpern, B., C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan and J. Whaley, *The Jalapeño virtual machine*, IBM Systems Journal **39** (2000), pp. 211–238.

[2] Amme, W., N. Dalton, M. Franz and J. von Ronne, *A typed-safe mobile code representation aimed at supporting dynamic optimization at the target side*, in: *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Monterey, CA, 2000, pp. 61–70.

[3] Amme, W., N. Dalton, J. von Ronne and M. Franz, *SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form*, in: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI 2001)* (2001), pp. 137–147.

[4] Arnold, M., S. Fink, D. Grove, M. Hind and P. F. Sweeney, *Adaptive optimization in the Jalapeno JVM*, in: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)* (2000), pp. 47–65.

[5] Burke, M. G., J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan and J. Whaley, *The jalapeño dynamic optimizing compiler for java*, in: *Proceedings of the ACM 1999 conference on Java Grande (Java 99)* (1999), pp. 129–141.

[6] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems (TOPLAS) **13** (1991), pp. 451–490.

[7] Hecht, M. S. and J. D. Ullman, *Characteristics of reducible flow graphs*, Journal of the ACM **21** (1974), pp. 367–375.

[8] Mathew, J. A., P. D. Coddington and K. A. Hawick, *Analysis and development of Java Grande Benchmarks*, in: *Proceedings of the ACM 1999 Java Grande Conference (Java 1999)* (1999), pp. 72–80.